Machine Learning Cloud Regression: The Swiss Army Knife of Optimization

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com

Version 1.0, August 2022

Abstract

This article is not about regression performed in the cloud. It is about considering your data set as a cloud of points or observations, where the concepts of dependent and independent variables (the response and the features) are blurred. It is a very general type of regression, offering backward-compatibility with existing methods. Treating a variable as the response amounts to setting a constraint on the multivariate parameter, and results in an optimization algorithm with Lagrange multipliers. The originality comes from unifying and bringing under a same umbrella, a number of disparate methods each solving a part of the general problem and originating from various fields. I also propose a novel approach to logistic regression, and a generalized R-squared adapted to shape fitting, model fitting, feature selection and dimensionality reduction. In one example, I show how the technique can perform unsupervised clustering, with confidence regions for the cluster centers obtained via parametric bootstrap.

Besides ellipse fitting and its importance in computer vision, an interesting application is non-periodic sum of periodic time series. While rarely discussed in machine learning circles, such models explain many phenomena, for instance ocean tides. It is particular useful in time-continuous situations where the error is not a white noise, but instead smooth and continuous everywhere. For instance, granular temperature forecast. Another curious application is modeling meteorite shapes. Finally, my methodology is model free and data driven, with a focus on numerical stability. Prediction intervals and confidence regions are obtained via bootstrapping. I provide Python code and synthetic data generators for replication purposes.

Contents

1 Introduction: circle fitting

The goal is to unify all regression techniques and present a simple, generic framework to solve most problems dealing with fitting an equation to a data set. Currently, there are dozens of types of regressions, each with its own methodology and algorithm. Here I propose a single methodology and a single algorithm to solve all these problems.

The originality of my technique resides in my approach and methodology, rather than in the type of math or algorithm being used. Like all generic methods, it is rather abstract and one would think more difficult to learn and describe. To the contrary, I believe it is actually more intuitive and easier to grasp. First, the dependent variable and independent features are interchangeable: the concept of dependent variables is even absent in my methodology. Thus I call it "cloud regression", as the data set is viewed as a cloud of points, with no particular axis or dimension being privileged unless explicitly required. Then the technique is model-free: it uses resampling and bootstrap to build prediction intervals, or confidence intervals for the regression coefficients.

A judicious choice of notations makes my methodology backward-compatible with all existing techniques. The concept of R-squared is slightly modified to offer extra possibilities: measuring the quality of the fit for the full model versus a sub-model of your choice. In standard regression, the sub-model is a constant and referred to as the base model. Here the sub-model could be fitting a circle if the full model is about ellipses, or fitting a plane versus an hyperplane in standard linear regression.

All regression books and chapters for beginners start with fitting a line. Here the easiest example – the first one to be taught – is fitting a circle centered at the origin. Think about it for a moment: intuitively, the estimated radius is the average radius computed on the data points. Indeed, this is the solution produced by my technique. The second easiest case is then fitting a line involving a slope and an intercept. Both examples are a particular case of fitting a quadratic form (ellipsoid).

This presentation is intended to beginners. There are examples, just as in standard regression, where the solution is not unique. In my opinion, non-uniqueness should be embraced rather than avoided: in real life one would expect that multiple, different shapes can fit to a particular data set. Finding several of them provides more insights about your data. However, conditions needed for uniqueness are not discussed here: this is the topic of a more advanced presentation.

In many cases, thanks to an appropriate re-parameterization, the solution is obtained using simple constrained optimization and Lagrange multipliers. It has more to do with elementary calculus than advanced matrix algebra. In particular, there is no explicit mention of eigenvalues [Wiki] or singular value decomposition [Wiki]. Also, the shape does not need to have derivatives, though if it does, a faster implementation is possible, with a Newton-like algorithm. Indeed, the shape may be differentiable nowhere: think about fitting a Brownian motion to a set of observations.

I provide examples using synthetic data [Wiki] and Python code. One of them involves time series forecasting with two periods p, q where p/q is not a rational number. Since p and q are among the parameters to be estimated, this is a true non-linear problem that can not be transformed into something linear via a link function [Wiki], unlike (say) logistic regression. A real life application, to benchmark the performance of the method, is predicting ocean tides: large, granular geospatial data sets are available to test the prediction algorithm in this non-linear context.

Finally, "cloud regression" encompasses the general linear model [Wiki], the generalized linear model [Wiki] (and thus logistic regression), as well as weighted least squares [Wiki] (and thus Bayesian regression). Via the mapping $z \mapsto w$ discussed in section 1.1, it can accommodate splines as in adaptive regression splines [Wiki]. Both cloud regression and the total least squares method [Wiki] minimize the sum of the squared distances between the data points and the shape, though my method does not give the response (called the dependent variable by statisticians) a particular status: in other words, it also works in the standard situation where there is no response, but just a cloud of points instead. In addition, my technique handles truly non-linear situations, unlike the generalized linear model. For that reason, I call it the mother of all regressions.

This is not the first time a regression technique does not discriminate between dependent and independent variables: partial least squares [Wiki] also allows for that. See also [9].

1.1 Previous versions of my method

The current version is much more general, simpler and radically different from the first implementation. However, it may help to provide some historical context. Initially, the goal was to compute the sum of the squared distances between a set of points (the observations, or the "cloud"), and a pre-specified shape Γ_θ (a line, plane or circle) belonging to a parametric family driven by a multidimensional parameter θ.

The idea was as follows. Let P_∞ be a fixed point located extremely far away from the shape, and P be a point of the training set. Draw the line L_P that goes through P and P_∞, and find the intersection $\Gamma_\theta \cap \text{Ł}_P$ closest to P, between the shape and the line. Let Q_θ be this point. The point in question may not be unique or may not exist (depending on θ), but the distance $\Delta_\theta(P) = ||P - Q_\theta||$ is, assuming there is an intersection. Then find θ^* that minimizes the sum of $\Delta_\theta(P)$ computed over all training set points P. This θ^*, if unique, determines the shape that best fits the data. Traditional projection-based techniques compute the exact distance between a point and a shape, and therefore require the shape to be differentiable. The method based on P_∞ works with shapes that are not differentiable. Some particular cases in the new implementation produce similar or identical results to those obtained with the P_∞ method.

If the shape in question is an hyperplane and the context is traditional multivariate linear regression, then the shape is defined by $g(w, \theta) = 0$ where $w = (y, x_1, \dots, x_m)$ and $g(w, \theta) = \theta_0 y + (\theta_1 x_1 + \cdots + \theta_m x_m)$. Here y corresponds to the dependent variable, $x_1, \dots, x_m$ to the features, and $\theta_0, \theta_1, \dots, \theta_m$ are the regression coefficients, with the constraint $\theta_0 = -1$. In the new methodology, the constraint $\theta_0 = -1$ is handled using a Lagrange multiplier, but other than that, it leads to the same traditional solution. If there is an intercept, then $x_1 = 1$. In the end, the goal is to propose a technique that is both general and intuitive, following the modern trend of explainable AI [Wiki].

In a second version of my methodology (not the current version), I introduced a mapping system, which essentially is a change of coordinates associated to a link function. The goal was to transform the data so that after the mapping, it is more amenable to a simple solution. Also, it is an attempt at obtaining a scale-independent solution: whether your unit is a mile or a kilometer should have no impact on the solution. In its most general form, the observations and parameters are denoted as z and φ. The shape satisfies the equation $h(z, \varphi) = 0$. The mapping is defined as $g(w, \theta) = \xi(h(z, \varphi))$ where $\xi : \mathbb{R} \to \mathbb{R}$ is a strictly monotonic function, with $w = \nu(z)$ and $\theta = \phi(\varphi)$, for some multivariate one-to-one mappings ν and ϕ. All the computations are done in the (w, θ)-space, thought it is possible to revert back to the original (z, φ) when computations are done, if ever needed.

I eventually dropped both ξ and simply ignored φ and ϕ, leading to a less abstract model, yet covering all practical cases. Thus in the current version, $h(z, \varphi) = g(w, \theta)$, and we don't care about φ. We may as well use $\varphi = \theta$. The mapping ν gives rise to spline regression in the new method. However, when splines are used, they are pre-specified rather than estimated, to avoid over-fitting. Typically, they are chosen to simplify the computations.

Finally, I was interested in some original dimension reduction technique. Not a true data reduction technique, but it allows you to reduce the number of parameters by a factor two: consider w and θ to be complex, rather than real numbers, for instance via a mapping $z \mapsto w$ from $\mathbb{R}^2$ to $\mathbb{C}$, with $w = \nu(z)$. A benefit is the possibility to use a conformal map [Wiki] for ν, thus preserving angles. Such an example is the log-polar map [Wiki] $z = \exp(w)$ with $g(w, \theta) = z^\theta = \exp(\theta w)$, which corresponds to using the polar coordinate system with $\theta, z, w \in \mathbb{C}$: it makes things easier when dealing with circular data. The next step was to look at quaternions to reduce the number of parameters by a factor four, but there are a number of challenges. Anyway, I promised to keep things simple in this introductory presentation, so I won't discuss complex or quaternion mappings here. This is the topic of future research.

It is interesting to note that the problem of circle fitting has been quite extensively studied, see [10]. Essentially, these methods solve the problem using φ and they are not trivial. The solution based on my method involves working with θ and leads to a very classic algorithm with a simple solution. The price to pay is that the θ parameters are less obvious to interpret: they are the coefficients of a quadratic form. To the contrary, the direct solution involves φ parameters that have obvious meaning: the radius of the circle, its center and (in case of an ellipse) the rotation angle. However, my approach makes it a lot easier to generalize to ellipses and even far more complicated shapes, or hyperplanes for that matter, while at the same time having a solution that is even simpler than those discussed in [10] and applicable to the circle only. Of course, in this case there is a one-to-one mapping between φ and θ, see here. So you can always retrieve φ from θ.

2 Methodology, implementation details and caveats

I encourage you to first read section 1.1, as it provides a good amount of context. This section describes the details of the methodology. For simplicity, I do not describe the most general case, but a case that is general enough to cover all practical applications. I start by introducing the concept of data (called point cloud), parameter, and shape.

The **data** set is denoted as W, and consists of $m+1$ variables and n observations. Thus W is a $n \times (m+1)$ matrix as usual. The k-th row corresponds to the k-th observation $W_k = (W_{k,0}, W_{k,1}, \dots, W_{k,m+1})$. For backward compatibility with traditional models, I use the notation $W_{k,0} = Y_k$ for the dependent variable or response (if there is one), and $(X_{k,1}, \dots, X_{k,m}) = (W_{k,1}, \dots, W_{k,m+1})$ for the independent variables of features. The column vector corresponding to the response is denoted as Y, and the $n \times m$ matrix representing the independent variables is denoted as X. The whole data set W is referred to as the point cloud.

The **parameter** is a multivariate column vector denoted as $\theta = (\theta_0, \theta_1, \dots, \theta_d)$, with $d+1$ components. Typically, $d = m$ and θ satisfies some constraint, specified by $\eta(\theta) = 0$ for some function η. The most common functions are $\eta(\theta) = \theta^T\theta - 1$, $\eta(\theta) = \theta_0 + 1$, and $\eta(\theta) = (\theta_0 + \cdots + \theta_d) - 1$. Here ^{T} denotes the matrix/vector transposition operator.

The purpose is to fit a **shape** to the point cloud. The most typical shapes, after proper mapping, are hyperplanes or quadratic forms (ellipsoids). The former is a particular case of the latter. The shape belongs to a parametric family of equations driven by the multivariate parameter θ. The equation of the shape is $g(w, \theta) = 0$, for some function g. Typical examples include $g(w, \theta) = w\theta$ and $g(w, \theta) = w\theta - 1$, with $d = m$. The former usually involves an intercept: $X_{k,1} = 1$ for all $k = 1, \dots, n$. Keep in mind that w and θ are vectors, but $g(w, \theta)$ is a real number, not a vector. Thus $w\theta$ represents a dot product [Wiki].

2.1 Solution, R-squared and backward compatibility

The shape that best fits the data corresponds to $\theta = \theta^*$, obtained by minimizing the squares:

$$\theta^* = \arg\min_{\theta} \sum_{k=1}^{n} g^2(W_k, \theta). \tag{1}$$

The solution may not be unique. Uniqueness and numerical stability will be addressed in a future article, but the basics are covered in this document. The constraint $\eta(\theta) = 0$ guarantees that the solution requires solving a (sometimes non-linear) system of $d + 2$ equations with $d + 2$ unknowns. In some cases, $d \leq m$ to avoid model identifiability issues [Wiki]. Also, a large d may result in overfitting [Wiki]. Then, you want $n > d$ otherwise the solution may not be unique unless you add more constraints on θ. The solution θ^* is obtained by solving the system

$$\begin{cases} \displaystyle\sum_{k=1}^{n} \nabla_\theta [g^2(W_k, \theta)] = \lambda \nabla_\theta [\eta(\theta)], \\ \eta(\theta) = 0 \end{cases} \tag{2}$$

where ∇_θ is the gradient operator with respect to θ [Wiki], and λ is called the Lagrange multiplier [Wiki]. This is a classic constrained convex optimization problem. The top part of (2) consists of a system of $d+1$ equations with $d + 2$ unknowns $\theta_0, \dots \theta_d$ and λ. The bottom part is a single equation with $d + 1$ unknowns $\theta_0, \dots \theta_d$. Combined together, it constitutes a system of $d + 2$ equations with $d + 2$ unknowns. Note the analogy with Lasso regression [Wiki] when $\eta(\theta) = \theta^T\theta - 1$, that is, when $\theta^T\theta = 1$.

The mean squared error (MSE) relative to a particular θ is defined as

$$\text{MSE}(\theta) = \frac{1}{n} \sum_{k=1}^{n} g^2(W_k, \theta) \geq \text{MSE}(\theta^*). \tag{3}$$

The inequality in (3) is an immediate consequence of (1). Now define the R-squared with respect to θ as

$$R^2(\theta) = 1 - \frac{\text{MSE}(\theta^*)}{\text{MSE}(\theta)}. \tag{4}$$

It follows immediately that $0 \leq R^2(\theta) \leq 1$. A perfect fit corresponds to $\text{MSE}(\theta^*) = 0$ (the whole cloud residing on the shape). In that case, if $\theta \neq \theta^*$ and the optimum θ^* is unique, then $R^2(\theta) = 1$.

In traditional linear regression, the R-squared is defined as $R^2(\theta_*)$ where θ_* is the optimum θ for the base model. The base model corresponds to all the coefficients θ_i attached to the independent variables set to zero, except the one attached to the intercept. In other words, in the base model, the predicted Y is constant, equal to the empirical mean of Y. As a result, $\text{MSE}(\theta_*) = \text{Var}[Y]$, the empirical variance of Y. A consequence is that $R^2(\theta_*)$ is the square of the correlation between the observed response Y, and the predicted response of the full model.

Backward compatibility with traditional linear regression works as follows. The standard univariate regression corresponds to $g(w, \theta) = w\theta = \theta_0 y + \theta_1 x + \theta_2$, with the constraint $\theta_0 = -1$. Thus $g(w, \theta) = 0$ if and only if $y = \theta_1 x + \theta_2$. This generalizes to multivariate regression as well. A more elegant formulation in the new methodology is to replace the constraint $\theta_0 = -1$ by the symmetric constraint $\theta_0^2 + \theta_1^2 + \theta_2^2 = 1$. Note that w is a row vector and θ is a column vector.

2.2 Upgrades to the model

By model, I mean the general setting of the method: there is no probabilistic model involved in this discussion. Prediction intervals [Wiki] for the individual error $g(W_k, \theta^*)$ at each data point W_k (or for the estimated response attached to Y_k, if there is an independent variable) and confidence regions [Wiki] for θ^* can be obtained via re-sampling and bootstrapping [Wiki]. This is also true for points outside the training set.

Also, the squares can be replaced by absolute values, as in quantile regression [Wiki], to minimize the impact of outliers and for scale preservation: if a variable is measured in years, then squares are expressed in squared years, a metric that is meaningless. This leads to a modified, better metric to assess the quality of the fit, replacing the R-squared. See the section "performance assessment" in my article "Little Known Secrets about Linear Regression" [4], for alternatives to the R-squared. The goodness of fit (say, the R-squared) should be measured on the validation set [Wiki] even though θ^* is computed on a subset of the training set: this is a standard practice, called cross-validation [Wiki], illustrated on synthetic data in my article on fuzzy regression [3].

Now, let's get back to the R-squared. In standard linear regression, the R-squared is defined as $R^2(\theta_*)$ via Formula (4), where θ_* is the optimum θ for the base model (the predicted response is constant, equal the mean of Y for the base model). In the new methodology, there may be no response. Still, the definition of R^2 extends to that situation, and is compatible with the traditional version. What's more, it leads to many possible R^2, one for each sub-model (not just the base model), and this is true too for the standard regression. A sub-model corresponds to adding constraints on the parameter vector θ, or in other words, working with a subset of the parameter space. Let θ_* be the optimum for a specific sub-model, while θ^* is the optimum for the full model. Then the definition of R^2, depending on θ_*, is unchanged. It could not be any simpler!

Now you can use R^2 for model comparison purposes and even for feature selection [Wiki]. You can test the improvement obtained by using the full model over a sub-model, with the metric $S(\theta_*) = R^2(\theta^*) - R^2(\theta_*)$. Here θ_* is the optimum θ attached to the sub-model. Obviously, $0 \leq S(\theta_*) \leq 1$. The larger $S(\theta_*)$, the bigger the improvement. Conversely, the smaller, the better the performance of the sub-model. Examples include fitting an ellipse (full model) versus fitting a circle (sub-model) or using all the features (full model) versus using a subset (sub-model). You can compare sub-models and rank them according to $S(\theta_*)$. This allows you to identify the smallest set of features that achieve a good enough $S(\theta_*)$, for dimensionality reduction purposes [Wiki].

Finally, another update consists of using positive weights $\psi_k(\theta)$ in Formula (1). This amounts to performing weighted regression [Wiki]. For instance, data points far away from the optimum shape, that is observations with a large $g^2(W_k, \theta^*)$, may be discarded to reduce the impact of outliers. Or the weights can be used to balance the coefficients θ_i, in an effort to achieve scale-invariance in the expression $w\theta$. Then the top system in (2) becomes

$$\sum_{k=1}^{n} \psi_k(\theta)\nabla_\theta[g^2(W_k,\theta)] + \sum_{k=1}^{n} g^2(W_k,\theta)\nabla_\theta[\psi_k(\theta)] = \lambda\nabla_\theta[\eta(\theta)]. \tag{5}$$

3 Case studies

In section 3.1, I show how to solve the logistic regression. The first version is standard least squares, to further illustrate backward compatibility with the traditional method. The second one illustrates how it could be done if you want to follow the spirit of the new methodology. Then I discuss two fundamental examples based on synthetic data.

3.1 Logistic regression, two ways

In the traditional setting, $w = (y, x)$ where y is the response, and x the features. For the logistic regression [Wiki], we have

$$g(w,\theta) = g(y,x) = y - F(x\theta), \quad \text{with } \ x\theta = \theta_1 x_1 + \dots \theta_m x_m.$$

Here $x_1 = 1$ corresponds to the intercept, thus we have $m-1$ actual features $x_2, x_3, \dots, x_m$. There is no constraint on the parameter θ, thus there is no function $\eta(\theta)$. In Formula (2), $\eta(\theta) = 0$ should be ignored, and $\lambda = 0$. The function F is a cumulative distribution function with a symmetric density around the origin. In this case, $F(x\theta) = 1/(1+\exp[-x\theta])$ is the standard logistic distribution [Wiki].

In the new methodology, one would proceed as follows. First, the original data is denoted as $z = (v, u)$. The logistic regression applies to the original data. Here v is the response, and u the feature vector. The parameter θ is unchanged (not subject to a mapping), and still denoted as θ. This regression can be stated as

$$g(z,\theta) = g(v,u) = v - F(u\theta), \quad \text{with } \ u\theta = \theta_1 u_1 + \dots \theta_m u_m.$$

The first step is to map $z = (v, u)$ onto $w = (y, x)$, with the hope of simplifying the problem, as discussed in section 1.1. This is done via the link function $y = F^{-1}(v) = \log[v/(1-v)]$ and $u = x$. Now we are back to

$$g(z,\theta) = g(w,\theta) = g(y,x;\theta) = y - x\theta, \quad \text{with } \ x\theta = \theta_1 x_1 + \dots \theta_m x_m.$$

This is how standard linear regression is expressed in the new framework. But it is still the traditional linear regression, with nothing new. The final step consists in extending θ, adding one component θ_0 to $\theta_1, \dots, \theta_m$. With the new θ (still denoted as θ) we have $g(w, \theta) = w\theta = \theta_0 w_0 + \cdots + \theta_m w_m$. You need to add one constraint on θ. The constraint $\theta_0 = -1$, that is $\eta(\theta) = \theta_0 + 1$, yields the exact same solution as traditional linear regression. But $\theta^T\theta = 1$, that is $\eta(\theta) = \theta^T\theta - 1$, makes the problem somehow symmetric, and more elegant.

However, in many applications, the response v in the original space is either 0 or 1, such as cancer versus non-cancer, or fraud versus non-fraud. In this case, the link function is undefined. The mapping with the link function works if the response is a proportion, strictly between zero and one. Otherwise, the standard logistic regression is the best approach. A possible workaround is to use for F a distribution with a finite support, such as uniform on $[a, b]$. Afterall, the observed values (the features) are always bounded anyway. Then, intuitively, given θ, estimates of a and b are proportional respectively to the minimum and maximum of $U_k\theta$, over $k = 1, \dots, n$.

This suggests a new approach to logistic regression. First, use the model $v = F_\theta(u\theta)$ in the (v, u)-space, where $0 \leq v \leq 1$ and F_θ is the empirical distribution [Wiki] of $u\theta$ given θ. Then choose θ^* that minimizes the sum of squared residuals:

$$\theta^* = \arg\min_\theta \sum_{k=1}^n g^2(V_k, U_k; \theta) = \arg\min_\theta \sum_{k=1}^n (V_k - F_\theta(U_k\theta))^2.$$

Remember, U_k is a row vector, and θ is a column vector; the dot product $U_k\theta$ is a real number. Also, V_k is the binary response attached to the k-th observation, while U_k is the corresponding m-dimensional feature vector, both in the original (v, u)-space. The empirical distribution F_θ is computed as follows: $F_\theta(t)$ is the proportion of observed feature vectors, among $U_1, \dots, U_n$, satisfying $U_k\theta \leq t$. Such a method could be called CDF regression. You can use the methodology presented here to solve it, but it would be very computer-intensive, because F_θ depends on θ in a non-obvious way. The predicted value for V_k, is $F_{\theta^*}(U_k\theta^*)$ in this case.

3.2 Ellipsoid and hyperplane fitting

This is a fundamental example, with hyperplanes being a particular case of ellipsoids. I illustrate the methodology with an example based on synthetic data, in a small dimension. The idea is to represent the shape with a quadratic form. In two dimensions, the equations is

$$\theta_0 x^2 + \theta_1 xy + \theta_2 y^2 + \theta_3 x + \theta_4 y + \theta_5 = 0.$$

The trick is to re-write it with artificial variables $w_0 = x^2, w_1 = xy, w_2 = y^2, w_3 = x, w_4 = y, w_5 = 1$, so that we can use the general framework with $g(w, \theta) = w\theta$. Again, $w\theta$ is the dot product. To avoid the trivial solution $\theta^* = 0$, let's add the constraint $\theta^T\theta = 1$, that is, $\eta(\theta) = \theta^T\theta - 1$. Then, θ^* is solution of the system

$$\begin{cases} (W^TW - \lambda I)\theta = 0, \\ \qquad\qquad \theta^T\theta = 1. \end{cases} \tag{6}$$

The above solution is correct in any dimension. It is a direct application of (2). Here W is the $n \times 6$ matrix containing the n observations. Thus, $W_{k0} = X_k^2, W_{k1} = X_kY_k, W_{k2} = Y_k^2, W_{k3} = X_k, W_{k4} = Y_k, W_{k5} = 1$. The Python code and additional details, for a slightly different version with a slightly different $\eta(\theta)$, can be found here. I use it in my own code, available on my GitHub repository, here, under the name `fittingEllipse.py`. It is based on Halir's article about fitting ellipses [7].

The Python code checks if the fitted shape is actually an ellipse. However, in the spirit of my methodology, it does not matter if it is an ellipse, a parabola, an hyperbola or even a line. The uniqueness of the solution is unimportant: indeed, if two very different solutions (say an ellipse and a parabola) yield the same minimum mean squared error and are thus both optimal, it says something about the data set, something interesting to know. However, it would be interesting to compute $R^2(\theta_*)$ using Formula (4), where θ_* corresponds to a circle. It would tell whether the full model (ellipse) over a significant improvement over the circle sub-model.

Ellipsoid fitting shares some similarities with multivariate polynomial regression [11]. The differences are:

- Ellipse fitting is a "full" model; in the polynomial regression $y = \theta_1 + \theta_2 x + \theta_3 x^2$, the terms y^2 and xy are always missing.
- Polynomial regression fits a curve that is unbounded such as $y = x^2$, resulting in poor fitting; to the contrary in ellipse fitting (if the solution is actually an ellipse) the solution is bounded.
- To get as many terms in polynomial regression as in ellipse fitting, the only way is to increase the degree of the polynomial, which further increases the instability of the solution.

Finally, Umbach [10] proposes a different approach to ellipse fitting. It is significantly more complicated, and indeed, they stopped at the circle. In short, their method directly estimates the center, semi-axis lengths and rotation angle via least squares, as opposed to estimating the coefficients in the quadratic form that represents the ellipse. More on parametric curves can be found in my article on shape recognition [1].

3.2.1 Curve fitting: 250 examples in one video

Ellipse fitting is performed by setting `mode='CurveFitting'` in the Python code. The program automatically creates a number of ellipses, specified by their parameters (center, lengths of semi-axes, and rotation angle), then generates a different training set for each ellipse, and outputs the result of the fitting procedure as an image. The images are then bundled together to produce a video, and an animated gif. Each image features a particular ellipse and training set, as well as the fitted ellipse based on the training set. The ellipses parameters are set by the variable `params` in the code: it is an array with five components. The number of ellipses is set by the parameter `nframes`, which also determines the number of frames in the output video.

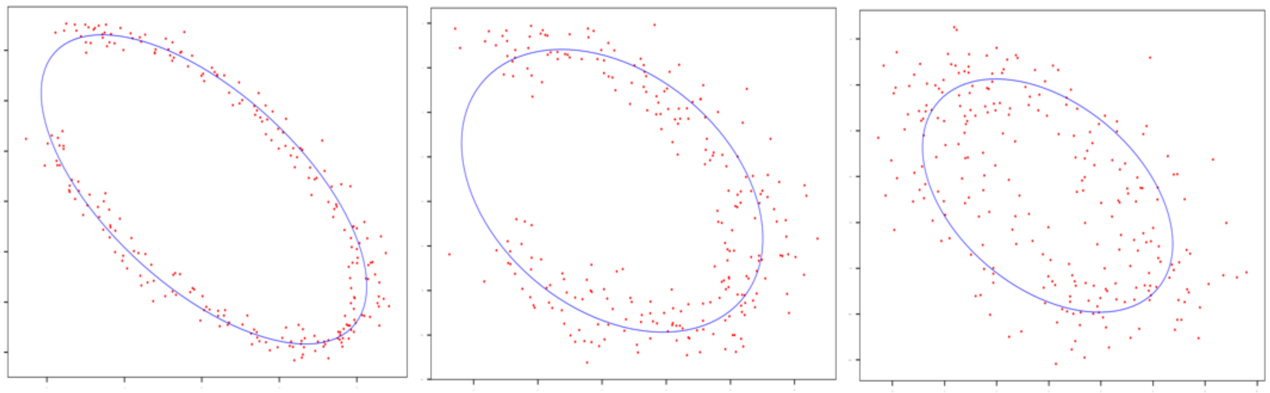


Figure 1: Fitted ellipse (blue), given the training set (red) distributed around a partial arc

Actually, the program does a little more: it works with ellipse arcs. Using the centroid of the training set to estimate the center of the ellipse does not work in this case. So the program retrieves the original, unknown ellipse even if the training set consists of points randomly distributed around a portion of that ellipse. The arc in question is determined by a lower and upper angle in polar coordinates, denoted respectively as `tmin` and `tmax` in the code, with `tmin=0` and `tmax=`2π corresponding to the full ellipse.

The training set consists of n observations generated as follows. First sample n points on the ellipse (or the arc you are interested in). Then perturb these points by adding some noise. You have two options: `noise_CDF='Uniform'` and `noise_CDF='Normal'`. The amount of noise is specified by the parameter `noise` in the code. For the uniform distribution on the square $[-a, a] \times [-a, a]$, `noise` represents a. For the bivariate normal distribution with covariance matrix $\sigma^2 I$ where I is the identity matrix, it represents σ^2. There are various ways of sampling points on an ellipse. Three options are offered here, set by the parameter `sampling`. They are described in section 3.2.3, in the paragraph "Sampling points on an ellipse arc". The option `'Enhanced'` is the only one performing stochastic sampling (points randomly picked up on the ellipse), and used in Figure 2.

In Figure 2, the size of the training set is $n = 30$ while in Figure 1, $n = 250$. In the code, n is represented by the variable `npts`. The training set is colored in red, the fitted ellipse in blue, and if present on the image as in Figure 2, the true ellipse is in black. The latter appears as a polygon rather than an ellipse because the sampled points on the true ellipse are joined by segments, and n is small. Typically, the true and fitted ellipses are very close to each other, although there is a systematic bias too small to be noticed to the naked eye unless the ellipse eccentricity is high. More on this soon.

Table 1 compares the exact parameter values (set by the user) of the true ellipse in Figure 2, to a few sets of estimated values obtained by least squares. Each set of estimates is computed using a different training set. All training sets are produced with same amount and type of noise, to give an idea of the variance of the parameter estimates at a specific level of noise. The five parameters are the ellipse center (x_0, y_0), the lengths of the semi axes (a_p, b_p), and the ellipse orientation (the rotation angle ϕ).

In some cases, the solution may not be unique, or could be an hyperbola or parabola rather than an ellipse. For instance, if the ellipse is reduced to a circle, any value for the rotation angle is de facto correct, though the estimated curve is still unique and correctly identified. Also, if the true ellipse has a high eccentricity, the generated white (unbiased) noise forces the training set points inside the ellipse more often than they should, as opposed to outside the boundary. This is because inside the ellipse, the noise from the North side strongly overlaps with the noise from the South side, assuming the long axis is the horizontal one. The result is biased estimates for a_p and b_p, smaller than the actual ones. In the end, the fitted curve has a higher eccentricity than the true one. The effect is more pronounced the higher the eccentricity. If the variance of the noise is small enough, there is almost no bias.

I posted a video featuring 250 fitted ellipses with the associated training sets, here on YouTube. It is also on GitHub, here. The accompanying animated gif is also on GitHub, here. All were produced with the Python code. In the video, the transition from one ellipse to the next one is very smooth. While I use 250 different combinations of arcs, rotation angles, eccentricities and noises to feature a large collection of very different cases, these configurations slowly progress from one frame to the next one in the video. But the 250 frames eventually cover a large spectrum of situations. The last one shows a perfect fit, where the training set points are all on the true ellipse.

3.2.2 Confidence region for the fitted ellipse

The computation of confidence regions is performed by setting `mode='ConfidenceRegion'` in the Python code. This time the program automatically creates a number of training sets (determined by the parameter `nframes`), for the same ellipse specified by its parameters `params`: center, lengths of semi-axes, and rotation angle. Then it estimates the ellipse parameters, and thus the true ellipse, uniquely determined by the parameters in question. Figure 2 shows the confidence region for the example outlined in Table 1.

	x_0	x_1	a_p	b_p	ϕ
Exact values	3.00000	−2.50000	7.00000	4.00000	0.78540
Training set 1	2.61951	−2.41818	6.44421	3.82838	0.72768
Training set 2	2.77270	−2.32346	6.59185	4.24624	0.59971
Training set 3	3.29900	−2.60532	6.71834	4.15181	0.87760
Training set 4	2.71936	−2.42349	7.15562	4.52900	0.80404

Table 1: Estimated ellipse parameters vs true values ($n = 30$), for shape in Figure 2

Actually I decided to display a polygon instead of the fitted ellipse, by selecting the option `sampling='Enhanced'`. The polygon consists of the predicted locations of the $n = 30$ training set points on the fitted ellipse. These locations are obtained in the exact same way that predicted values are obtained in a linear regression problem and then shown on the fitted line. After all, ellipse fitting as presented in this article is a particular case of the general cloud regression technique. I then joined these points using segments, resulting in one polygon per training set. The superimposition of these polygons is the confidence region.

The reason for using polygons rather than ellipses is for a particular application: estimating the shape of a small, far away celestial body based on a low resolution image. This is particularly useful when creating a taxonomy of these bodies: the shape parameters are used to classify them and understand their history as well as gravity interactions, and can be used as features in a machine learning algorithm. Then, for a small meteorite, people expect to see it as a polyhedron (the 3D version of a polygon) rather than an ellipsoid. Of course, if the number n of points in the training set is large, then the polyhedron is indistinguishable from the fitted ellipsoid. But in practice, with low resolution images, n is usually pretty small.

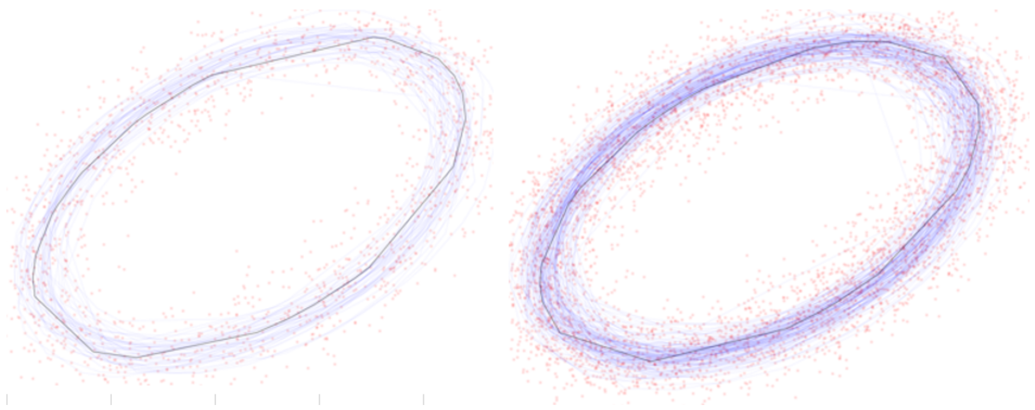


Figure 2: Confidence region in blue, $n = 30$ training set points; 50 training sets (left) vs 150 (right)

3.2.3 Python code

The main parameters in the code are highlighted in red in this high level summary. The program is listed in this section and also available on GitHub here, under the name `fittingEllipse.py`.

The least square optimization is performed using an implementation of the Halir and Flusser algorithm [7], adapted from a version posted here by Christian Hill, the author of the book “Learning Scientific Programming with Python” [8]. The optimization – minimizing the sum of squared errors between the observed points and the fitted ellipse – is performed on the coefficients of the quadratic equation representing the ellipse. This is the easiest way to do it, and it is also the approach that I use elsewhere in this article. The function `fit_ellipse` does the job, while `cart_to_pol` converts these coefficients into meaningful features: the center, rotation angle, eccentricity and the major and minor semi-axes of the ellipse [Wiki].

Sampling points on an ellipse arc

The Python code also integrates other components written by various authors. First, it offers three options to sample points on an ellipse or a partial arc of an ellipse, via the parameter `sampling` in the main section of the code:

- Evenly spaced on the perimeter, via the function `sample_from_ellipse_even`. The code is adapted from an anonymous version posted here. It requires the evaluation of elliptic integrals [Wiki]. The technique is identical to that described in the section “weighted centroid” in my article on shape detection [1].
- Randomly chosen on the perimeter, in such a way that on average, the distance between two consecutive sampled points on the ellipse is constant. It involves sampling from a multinormal distribution, rescaling the points and then sorting the sampled points so that they are ordered on the perimeter. This also requires sorting an array according to another array. It is done in the function `sample_from_ellipse`.
- The standard, easiest but notoriously skewed sampling. It consists of choosing equally spaced angles in the polar representation of the ellipse. For curve fitting, it is good enough with very little differences compared to the two other methods.

For sampling on a partial arc rather then the full ellipse, set the parameters `tmin` and `tmax` to the appropriate values, in the main loop. These are angles in the polar coordinate system, and should lie between 0 and 2π. The full ellipse corresponds to `tmin` set to zero, and `tmax` set to 2π.

Training set and ellipse parameters

Then, to create the training set, perturb the sampled points on the ellipse via uniform or Gaussian noise. The choice is set by the parameter `noise_CDF` in the main section of the code. The parameter `noise` determines the amount of noise, or in other words, the noise variance. Points, be it on the ellipse or in the training set, are arrays with names `x` and `y` (respectively for the X and Y coordinates). The number of points in the training set is determined by the parameter `npts`.

The shape of the ellipse is set by the 5-dimensional parameter vector `params`. Its components, denoted as `x0`, `y0`, `ap`, `bp`, `phi` throughout the code, are respectively the center of the ellipse, the length of the semi-axes, and the orientation of the ellipse (the rotation angle).

Confidence regions versus curve fitting

The program creates `nframes` ellipses, one at a time in the main loop. At each iteration, the created ellipse and training set is saved as a PNG image, for inclusion in a video or animated gif (see next paragraph). This is why the variable controlling the main loop is called `frame`. At each iteration the true parameters of the ellipse (the ones you chose), and their least squares estimates are displayed on the screen.

If the parameter `mode` is set to `'ConfidenceRegion'`, then the amount of noise and all ellipse parameters are kept constant throughout the iterations. The fitted shapes varies from one iteration to the next depending on the training set (itself depending on the noise), creating a confidence region for a specific ellipse, given a specific amount of noise. New fitted ellipses keep being added to the image without erasing older ones, to display the confidence region under construction. Highly eccentric ellipses result in biased confidence regions. The method used to build the confidence region is known as parametric bootstrap [Wiki].

To the contrary, if `mode` is set to `'FittingCurves'`, a different ellipse with different parameters and different amount of noise is generated at each iteration, erasing the previous one in the new image. The purpose in this case is to assess the quality of the fit depending on the amount of noise and the shape of the ellipse (the eccentricity and whether you use a full or partial arc for training, in particular).

Creating videos and animated gifs

At each iteration in the main loop, the program creates and saves an image in your local folder, featuring the training set in red (a cloud of dots distributed around the true ellipse arc) and the fitted ellipse in blue. The name of the image is `ellipsexxx.png`, where `xxx` is the current frame number. At the last iteration (the last frame in the video), the true ellipse – the one with the parameters set in the main loop – is added to the image, in black: it allows you to assess the bias when choosing the option `mode='ConfidenceRegion'`.

The video is saved as `ellipseFitting.mp4`, and the animated gif as `ellipseFitting.gif`. The parameter `DPI` (dots per inch) sets the resolution of the images. For videos, I recommend to set it to 300. For animated gifs, I recommend using 100. At the bottom of the code, when creating the video with a Moviepy function, you are free to change `fps=20` to any other value. This parameter sets the number of frames per second. Color transparency [Wiki] is used throughout the plots, to improve the rendering when multiple curves overlap. The transparency level is denoted as `alpha` in the code. You are not supposed to play with it unless you don't like my choice. I mention it just in case you are wondering what `alpha` represents.

Finally, if the parameter `ShowImage` is set to `True`, each frame is also displayed on your screen. The default value is `False`. Turn it on only if you produce a very small number of frames, say `nframes=10` or less.

```
import numpy as np
import matplotlib.pyplot as plt
import moviepy.video.io.ImageSequenceClip # to produce mp4 video
from PIL import Image # for some basic image processing

def fit_ellipse(x, y):

   # Fit the coefficients a,b,c,d,e,f, representing an ellipse described by
   # the formula F(x,y) = ax^2 + bxy + cy^2 + dx + ey + f = 0 to the provided
   # arrays of data points x=[x1, x2, ..., xn] and y=[y1, y2, ..., yn].

   # Based on the algorithm of Halir and Flusser, "Numerically stable direct
   # least squares fitting of ellipses'.

   D1 = np.vstack([x**2, x*y, y**2]).T
   D2 = np.vstack([x, y, np.ones(len(x))]).T
   S1 = D1.T @ D1
   S2 = D1.T @ D2
   S3 = D2.T @ D2
   T = -np.linalg.inv(S3) @ S2.T
   M = S1 + S2 @ T
   C = np.array(((0, 0, 2), (0, -1, 0), (2, 0, 0)), dtype=float)
   M = np.linalg.inv(C) @ M
   eigval, eigvec = np.linalg.eig(M)
   con = 4 * eigvec[0]* eigvec[2] - eigvec[1]**2
   ak = eigvec[:, np.nonzero(con > 0)[0]]
   return np.concatenate((ak, T @ ak)).ravel()

def cart_to_pol(coeffs):

   # Convert the cartesian conic coefficients, (a, b, c, d, e, f), to the
   # ellipse parameters, where F(x, y) = ax^2 + bxy + cy^2 + dx + ey + f = 0.
   # The returned parameters are x0, y0, ap, bp, e, phi, where (x0, y0) is the
   # ellipse centre; (ap, bp) are the semi-major and semi-minor axes,
   # respectively; e is the eccentricity; and phi is the rotation of the semi-
   # major axis from the x-axis.

   # We use the formulas from https://mathworld.wolfram.com/Ellipse.html
   # which assumes a cartesian form ax^2 + 2bxy + cy^2 + 2dx + 2fy + g = 0.
   # Therefore, rename and scale b, d and f appropriately.
   a = coeffs[0]
   b = coeffs[1] / 2
   c = coeffs[2]
   d = coeffs[3] / 2
   f = coeffs[4] / 2
   g = coeffs[5]

   den = b**2 - a*c
   if den > 0:
      raise ValueError('coeffs do not represent an ellipse: b^2 - 4ac must'
                       ' be negative!')
```

```
    # The location of the ellipse centre.
    x0, y0 = (c*d - b*f) / den, (a*f - b*d) / den

    num = 2 * (a*f**2 + c*d**2 + g*b**2 - 2*b*d*f - a*c*g)
    fac = np.sqrt((a - c)**2 + 4*b**2)
    # The semi-major and semi-minor axis lengths (these are not sorted).
    ap = np.sqrt(num / den / (fac - a - c))
    bp = np.sqrt(num / den / (-fac - a - c))

    # Sort the semi-major and semi-minor axis lengths but keep track of
    # the original relative magnitudes of width and height.
    width_gt_height = True
    if ap < bp:
       width_gt_height = False
       ap, bp = bp, ap

    # The eccentricity.
    r = (bp/ap)**2
    if r > 1:
       r = 1/r
    e = np.sqrt(1 - r)

    # The angle of anticlockwise rotation of the major-axis from x-axis.
    if b == 0:
       phi = 0 if a < c else np.pi/2
    else:
       phi = np.arctan((2.*b) / (a - c)) / 2
       if a > c:
          phi += np.pi/2
    if not width_gt_height:
       # Ensure that phi is the angle to rotate to the semi-major axis.
       phi += np.pi/2
    phi = phi % np.pi

    return x0, y0, ap, bp, phi

def sample_from_ellipse_even(x0, y0, ap, bp, phi, tmin, tmax, npts):

    npoints = 1000
    delta_theta=2.0*np.pi/npoints
    theta=[0.0]
    delta_s=[0.0]
    integ_delta_s=[0.0]
    integ_delta_s_val=0.0
    for iTheta in range(1,npoints+1):
       delta_s_val=np.sqrt(ap**2*np.sin(iTheta*delta_theta)**2+ \
                      bp**2*np.cos(iTheta*delta_theta)**2)
       theta.append(iTheta*delta_theta)
       delta_s.append(delta_s_val)
       integ_delta_s_val = integ_delta_s_val+delta_s_val*delta_theta
       integ_delta_s.append(integ_delta_s_val)
    integ_delta_s_norm = []
    for iEntry in integ_delta_s:
       integ_delta_s_norm.append(iEntry/integ_delta_s[-1]*2.0*np.pi)

    x=[]
    y=[]
    for k in range(npts):
       t = tmin + (tmax-tmin)*k/npts
       for lookup_index in range(len(integ_delta_s_norm)):
          lower=integ_delta_s_norm[lookup_index]
          upper=integ_delta_s_norm[lookup_index+1]
          if (t >= lower) and (t < upper):
             t2 = theta[lookup_index]
             break
```

```
      x.append(x0 + ap*np.cos(t2)*np.cos(phi) - bp*np.sin(t2)*np.sin(phi))
      y.append(y0 + ap*np.cos(t2)*np.sin(phi) + bp*np.sin(t2)*np.cos(phi))

   return x, y

def sample_from_ellipse(x0, y0, ap, bp, phi, tmin, tmax, npts):

   x=np.empty(npts)
   y=np.empty(npts)
   x_unsorted=np.empty(npts)
   y_unsorted=np.empty(npts)
   angle=np.empty(npts)

   # sample from multivariate normal, then rescale
   cov=[[ap,0],[0,bp]]
   count=0
   while count < npts:
      u, v = np.random.multivariate_normal([0, 0], cov, size = 1).T
      d=np.sqrt(u*u/(ap*ap) + v*v/(bp*bp))
      u=u/d
      v=v/d
      t = np.pi + np.arctan2(-ap*v,-bp*u)
      if t >= tmin and t <= tmax:
         x_unsorted[count] = x0 + np.cos(phi)*u - np.sin(phi)*v
         y_unsorted[count] = y0 + np.sin(phi)*u + np.cos(phi)*v
         angle[count]=t
         count=count+1

   # sort the points x, y for nice rendering with mpl.plot
   hash={}
   hash = dict(enumerate(angle.flatten(), 0)) # convert array angle to dictionary
   idx=0
   for w in sorted(hash, key=hash.get):
      x[idx]=x_unsorted[w]
      y[idx]=y_unsorted[w]
      idx=idx+1

   return x, y

def get_ellipse_pts(params, npts=100, tmin=0, tmax=2*np.pi, sampling='Standard'):

   # Return npts points on the ellipse described by the params = x0, y0, ap,
   # bp, e, phi for values of the parametric variable t between tmin and tmax.

   x0, y0, ap, bp, phi = params

   if sampling=='Standard':
      t = np.linspace(tmin, tmax, npts)
      x = x0 + ap * np.cos(t) * np.cos(phi) - bp * np.sin(t) * np.sin(phi)
      y = y0 + ap * np.cos(t) * np.sin(phi) + bp * np.sin(t) * np.cos(phi)
   elif sampling=='Enhanced':
      x, y = sample_from_ellipse(x0, y0, ap, bp, phi, tmin, tmax, npts)
   elif sampling=='Even':
      x, y = sample_from_ellipse_even(x0, y0, ap, bp, phi, tmin, tmax, npts)

   return x, y

def vgplot(x, y, color, alpha, npts, tmin, tmax):

   plt.plot(x, y, linewidth=0.2, color=color,alpha=alpha) # plot exact ellipse
   # fill gap (missing segment in the ellipse plot) if plotting full ellipse
   if tmax-tmin > 2*np.pi - 0.01:
      gap_x=[x[npts-1],x[0]]
      gap_y=[y[npts-1],y[0]]
      plt.plot(gap_x, gap_y, linewidth=0.2, color=color,alpha=alpha)
   return()
```

```
def main(npts, noise, seed, tmin, tmax, params, sampling):

   # params = x0, y0, ap, bp, phi (input params for ellipse)

   # Get points x, y on the exact ellipse and plot them
   x, y = get_ellipse_pts(params, npts, tmin, tmax, sampling)
   if frame == nframes-1 and mode == 'ConfidenceRegion':
      vgplot(x, y,'black', 1, npts, tmin, tmax)

   # perturb x, y on the ellipse with some noise, to produce training set
   np.random.seed(seed)
   if noise_CDF=='Normal':
     cov = [[1,0],[0,1]]
     u, v = np.random.multivariate_normal([0, 0], cov, size = npts).T
     x += noise * u
     y += noise * v
   elif noise_CDF=='Uniform':
     x += noise * np.random.uniform(-1,1,size=npts)
     y += noise * np.random.uniform(-1,1,size=npts)

   # get and print exact and estimated ellipse params
   coeffs = fit_ellipse(x, y) # get quadratic form coeffs
   print('True ellipse : x0, y0, ap, bp, phi = %+.5f %+.5f %+.5f %+.5f %+.5f' % params)
   fitted_params = cart_to_pol(coeffs) # convert quadratic coeffs to params
   print('Estimated values: x0, y0, ap, bp, phi = %+.5f %+.5f %+.5f %+.5f %+.5f' %
       fitted_params)
   print()

   # plot training set points in red
   if mode == 'ConfidenceRegion':
     alpha=0.1 # color transparency for Confidence Regions
   elif mode == 'CurveFitting':
     alpha=1
   plt.scatter(x, y,s=0.5,color='red',alpha=alpha)

   # get points on the fitted ellipse and plot them
   x, y = get_ellipse_pts(fitted_params,npts, tmin, tmax, sampling)
   vgplot(x, y,'blue', alpha, npts, tmin, tmax)

   # save plots in a picture [filename is image]
   plt.savefig(image, bbox_inches='tight',dpi=dpi)
   if ShowImage:
      plt.show()
   elif mode=='CurveFitting':
      plt.close() # so, each video frame contains one curve only
   return()

#--- Main Part: Initializationa

noise_CDF='Normal' # options: 'Normal' or 'Uniform'
sampling='Enhanced' # options: 'Enhanced', 'Standard', 'Even'
mode='ConfidenceRegion' # options: 'ConfidenceRegion' or 'CurveFitting'
npts = 25          # number of points in training set

ShowImage = False # set to False for video production
dpi=100  # image resolution in dpi (100 for gif / 300 for video)
flist=[] # list of image filenames for the video
gif=[]   # used to produce the gif image
nframes=50 # number of frames in video

# intialize plotting parameters
plt.rcParams['axes.linewidth'] = 0.5
plt.rc('axes',edgecolor='black') # border color
plt.rc('xtick', labelsize=6) # font size, x axis
plt.rc('ytick', labelsize=6) # font size, y axis
```

```
#--- Main part: Main loop

for frame in range(0,nframes):

   # Global variables: dpi, frame, image
   image='ellipse'+str(frame)+'.png' # filename of image in current frame
   print("Creating image",image) # show progress on the screen

   # params = (x0, y0, ap, bp, phi) : first two coeffs is center of ellipse, last one
   # is rotation angle, the two in the middle are the semi-major and semi-minor axes

   if mode=='ConfidenceRegion':
      seed=frame  # new set of random numbers for each image
      noise=0.8   # amount of noise added to to training set
      # 0 <= tmin < tmax <= 2 pi
      tmin=0      # training set: ellipse arc starts at tmin
      tmax = 2*np.pi # training set: ellipse arc ends at tmax
      params = 3, -2.5, 7, 4, np.pi/4 # ellipse parameters
   elif mode=='CurveFitting':
      seed = 100     # same seed (random number generator) for all images
      p=frame/(nframes-1) # assumes nframes > 1
      noise=3*(1-p)*(1-p) # amount of noise added to to training set
      # 0 <= tmin < tmax <= 2 pi
      tmin= (1-p)*np.pi # training set: ellipse arc starts at tmin
      tmax= 2*np.pi  # training set: ellipse arc ends at tmax
      params = 4, -3.5, 7, 1+6*(1-p), 2*(p+np.pi/3) # ellipse parameters

   # call to main function
   main(npts, noise, seed, tmin, tmax, params, sampling)

   # processing images for video and animated gif production (using pillow library)
   im = Image.open(image)
   if frame==0:
     width, height = im.size # determines the size of all future images
     width=2*int(width/2)
     height=2*int(height/2)
     fixedSize=(width,height) # even number of pixels for video production
   im = im.resize(fixedSize) # all images must have same size to produce video
   gif.append(im)  # to produce Gif image [uses lots of memory if dpi > 100]
   im.save(image,"PNG") # save resized image for video production
   flist.append(image)

# output video / fps is number of frames per second
clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=20)
clip.write_videofile('ellipseFitting.mp4')

# output video as gif file
gif[0].save('ellipseFitting.gif',save_all=True, append_images=gif[1:],loop=0)
```

3.3 Non-periodic sum of periodic time series

In this section I consider the problem of fitting a non-periodic trigonometric series via least squares. One well known example is the Dirichlet eta function with an infinite number of superimposed periods. Its modulus is pictured in Figure 3. Practical application are also numerous. A good exercise is to download ocean tide data, and use the methodology described in this section to predict low and high tides, at various locations: the tides are influenced mostly by two factors – gravitation from the moon and from the sun – each with its own period. But the combination is not periodic. The fitting technique allows you to quantify the effect of each hidden component (the sun and the moon in this case) and retrieve their respective periods. The tide data is available for free, here.

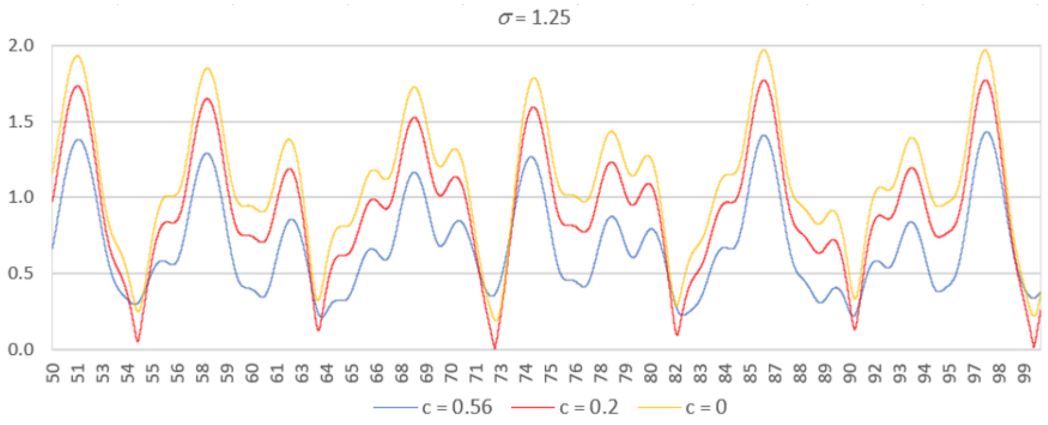


Figure 3: Three non-periodic time series made of periodic terms. Source: [5].

With my notation, the problem is defined by $w = (y, x)$ and

$$g(y, x; \theta) = y - \Big[\theta_1 \cos(\theta_2 x + \theta_3) + \theta_4 \cos(\theta_5 x + \theta_6)\Big]. \tag{7}$$

There is no constraint on θ, and thus no η function and no λ in (2). Here y is the response, and x represents the time. For this reason, I also use the notation $y = f_\theta(x)$, equivalent to $g(y, x; \theta) = 0$. This type of problem is called curve fitting [Wiki] in scientific computing. There are libraries to solve it: in Python, one can use the `optimize.curve_fit` function from the Scipy library. For more details, see the Python documentation.

Finally, if you are only interested in predicting y given x , but not in estimating the parameter vector θ, then the following interpolation formula is sometimes useful:

$$y = f(x) \approx \frac{\sin \pi x}{\pi} \cdot \left[\frac{f(0)}{x} + 2x \sum_{k=1}^{n} (-1)^k \frac{f(k)}{x^2 - k^2}\right] \tag{8}$$

I used it in Exercise 9 in my article on the Riemann Hypothesis [5], to get a good approximation of $y = f_\theta(x)$ when y is known (that is, observed) at integer increments $x = 0, 1$ and so on, even though θ is not known. I applied it to a function $f_\theta(x)$ closely related to those pictured in Figure 3. The function in question (namely, the real part of the Dirichlet eta function) can be expressed as an infinite sum of cosine terms with different amplitudes and different periods. Thus, in this case, the dimension of the unobserved θ is infinite, and θ remains an hidden parameter in the prediction experiment, hidden to the experimenter (as in a blind test). Its components are the various period and amplitude coefficients. The approximation formula (8) works under certain conditions: see Exercise 9 in [5] for details.

3.3.1 Numerical instability and how to fix it

Consider the simpler case where $\theta_3 = \theta_6 = 0$: let's drop these two coefficients from the model. Even then, The problem is ill-conditioned [Wiki]. In particular if $\theta^* = (\theta_1^*, \theta_2^*, \theta_4^*, \theta_5^*)$ is an optimum solution, so is $(\theta_4^*, \theta_5^*, \theta_1^*, \theta_2^*)$. At the very least, without loss of generality, you need to add the constraint $|\theta_1| \geq |\theta_4|$.

In Python, I used the `curve_fitting` function from Scipy. It does a poor job for this problem, even if you specify bounds for the coefficients $\theta_1, \theta_2, \theta_4, \theta_5$ and start the algorithm with a vector θ close to an optimum θ^*. My test involved

- Finding an optimum θ if the fitting function is $y = f_\theta(x) = \theta_1 \cos \theta_2 x + \theta_4 \cos \theta_5 x$,
- Using a synthetic training set where $y = a_1 \cos a_2 x + a_4 \cos a_5 x + a_7 \cos a_8 x$.

The gray curve in Figure 4, called the "model", is $y = a_1 \cos a_2 x + a_4 \cos a_5 x$, while the blue one is the fitted curve (not necessarily unique), and the dots represent the observations (training set in red, validation set in orange). The observations points do not lie exactly on the gray curve because I introduced some noise: the third term $a_7 \cos a_8 x$ between the model and the data. Note that the observations are equally spaced with respect to the X-axis, but absolutely not with respect to the Y-axis. It is possible to use a different sampling mechanism to address this issue. The figure was produced with the Python code in section 3.3.2. The values of $a_1, \ldots, a_8$ are pre-specified. Evidently, if $a_7 = 0$, then an obvious optimum solution is $\theta_i^* = a_i$ for $i = 1, 2, 4, 5$. It provides a perfect fit. Also, the coefficient a_7 specifies the amount of noise in the data.

Unfortunately, `curve_fitting` fails or performs very poorly in most cases. Figure 4 shows one of the relatively rare cases where it works well in the presence of noise. This Python function is still very useful in many contexts, but not in our example. The default setting (`method='lm'`, to be avoided) uses a supposedly robust version of the Levenberg-Marquardt algorithm, dating back to 1977, see here. Essentially, it gets stuck in local minima or fails to converge, and may even reject a manually chosen initial condition close to an optimum as “not feasible”. Surprisingly, increasing the amount of noise in the data, can provide improvements.

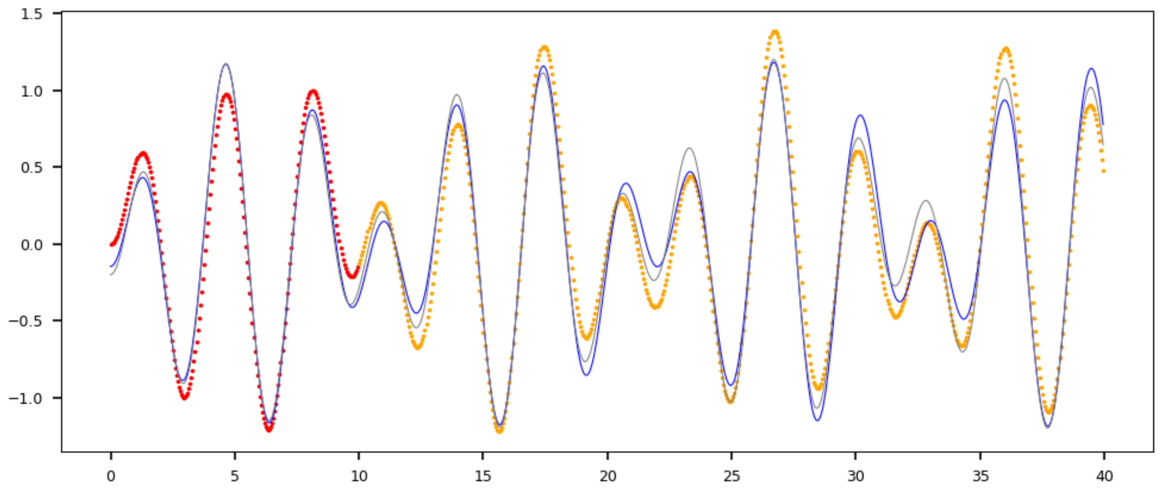


Figure 4: Training set (red), validation set (orange), fitted curve (blue) and model (gray)

To fix the numerical instability problem, one can use a more modern technique, such as swarm optimization [Wiki]. The `PySwarms` library documented here is the Python solution. For a tutorial, see here. A simpler home-made solution taking advantage of the fact that the fitting curve $f_\theta(x)$ (called regression function by statisticians) is linear both in θ_1 and θ_4, consists in splitting the problem as follows.

- Step 1: Sample (θ_2, θ_5) over a region large enough to encompass the optimum values.
- Step 2: Given θ_2, θ_5, find θ_1, θ_4 that minimizes $E(\theta) = \sum_{k=1}^{n} g^2(Y_k, X_k; \theta)$.

Here (Y_k, X_k) is the k-th observation in the training set, and $\theta = (\theta_1, \theta_2, \theta_4, \theta_5)$. Both steps are straightforward. You repeat them until you reach a point where the minimum $E(\theta)$ computed over the past iterations almost never decreases anymore. Step 2 is just a simple standard two-dimensional linear regression with no intercept, with an exact solution. One of the benefits is that if there are multiple minima, you are bound to find them all, without facing convergence issues. It also reduces a 4-D Monte-Carlo simulation to a 2-D one.

3.3.2 Python code

Despite the issues previously described, I decided to include the Python code. It shows how the `curve_fitting` function works, beyond using the default settings. It is still a good optimization technique for many problems such as polynomial regression. Unfortunately, not for our problem. If you decrease the amount of noise from `a7=0.2` to `a7=0.1` in the code below, there is a dramatic drop in performance. Likewise, if you change the initial θ_5 (the last component in `θ_start`) from 1.8 to 1.7, the performance collapses. The exact value in this example is $\theta_5 = 2$ by construction; the estimated (final) value produced by the Python code is about 1.995.

	θ_1	θ_2	θ_4	θ_5
Start	0.00000	1.00000	0.00000	1.80000
End	0.52939	1.42184	−0.67571	1.99526
Model	0.50000	1.41421	−0.70000	2.00000

Table 2: First and last step of `curve_fitting`, approaching the model.

Table 2 shows the quality of the estimation, for the parameter vector $\theta = (\theta_1, \theta_2, \theta_4, \theta_5)$. The `curve_fitting` procedure starts with an initial guess `θ_start` labeled “Start” in the table, and ends with the entry marked as “End” in the table: supposedly, close to an optimum θ^*. Because of the way the synthetic data is generated (within the Python code), the row marked “Model” and consisting of the value a_1, a_2, a_4, a_5 is always close to an optimum θ^*, unless the amount of noise introduced in the training set is too large. The “End” solution (the output of `curve_fitting`) is based exclusively on the training set points (the red dots in Figure 4), not on the validation set (the orange dots). Yet the approximation is unusually good, given the amount of noise.

By noise, I don't mean a random or Gaussian noise. Here the noise is deterministic: the purpose of this test is to check how well we can predict a phenomena modeled by a superimposition of multiple cosine terms of arbitrary periods, phases and amplitudes – for instance ocean tides over time – if we only use a sum of two cosine terms as an approximation. This model (its generalization with more terms) is particular useful in situations where the error is not a white noise [Wiki], but instead smooth and continuous everywhere: for instance in granular temperature forecast.

The curve fitting code, also producing Figure 4, is on my GitHub repository, here, under the name `fittingCurve.py`, and also listed below. I use Greek letters in the code to represent the θ vector and its components, for consistency reasons. Python digests them with no problem.

```
import numpy as np
import matplotlib as mpl
from scipy.optimize import curve_fit
from matplotlib import pyplot, rc

# initializations, define functions

def fpred(x, θ1, θ2, θ4, θ5):
  y = θ1*np.cos(θ2*x)+ θ4*np.cos(θ5*x)
  return y

def fobs(x, a1, a2, a4, a5, a7, a8):
  y = a1*np.cos(a2*xobs)+a4*np.cos(a5*xobs)+a7*np.cos(a8*xobs)
  return y

n=800
n_training=200 # first n_training points is training set
x=[]
y_obs=[]
y_pred=[]
y_exact=[]

# create data set (observations)

a1=0.5
a2=np.sqrt(2)
a4=-0.7
a5=2
a7=0.2 # noise (e=0 means no noise)
a8=np.log(2)

for k in range(n):
  xobs=k/20.0
  x.append(xobs)
  y_obs.append(fobs(xobs, a1, a2, a4, a5, a7, a8))

# curve fitting between f and data, on training set

θ_bounds=((-2.0, -2.5, -1.0, -2.5),(2.0, 2.5, 1.0, 2.5))
θ_start=(0.0, 1.0, 0.0, 1.8)
popt, _ = curve_fit(fpred, x[0:n_training], y_obs[0:n_training],\
   method='trf',bounds=θ_bounds,p0=θ_start)
θ1, θ2, θ4, θ5 = popt
print('Estimates  : θ1=%.5f θ2=%.5f θ4=%.5f θ5=%.5f' % (θ1, θ2, θ4, θ5))
print('True values: θ1=%.5f θ2=%.5f θ4=%.5f θ5=%.5f' % (a1, a2, a4, a5))
print('Initial val: θ1=%.5f θ2=%.5f θ4=%.5f θ5=%.5f' % \
   (θ_start[0], θ_start[1], θ_start[2], θ_start[3]))

# predictions

for k in range(n):
  xobs=x[k]
  y_pred.append(fpred(xobs, θ1, θ2, θ4, θ5))
  y_exact.append(fpred(xobs, a1, a2, a4, a5))
```

```
# show plot

mpl.rcParams['axes.linewidth'] = 0.5
rc('axes',edgecolor='black') # border color
rc('xtick', labelsize=6) # font size, x axis
rc('ytick', labelsize=6) # font size, y axis
pyplot.scatter(x[0:n_training],y_obs[0:n_training],s=0.5,color='red')
pyplot.scatter(x[n_training:n],y_obs[n_training:n],s=0.5,color='orange')
pyplot.plot(x, y_pred, color='blue',linewidth=0.5)
pyplot.plot(x, y_exact, color='gray',linewidth=0.5)
pyplot.show()
```

3.4 Fitting a line in 3D, unsupervised clustering, and other generalizations

In three dimensions, a line is the intersection of two planes A and B, respectively with equations $g_1(w, \theta_A) = 0$ and $g_1(w, \theta_B) = 0$. For instance, $g_1(w, \theta_A) = \theta_0 w_0 + \theta_1 w_1 + \theta_2 w_2 - \theta_3$. To fit the line,

- let $\theta_A = (\theta_0, \theta_1, \theta_2, \theta_3)^T$ and $\theta_B = (\theta_4, \theta_5, \theta_6, \theta_7)^T$,
- use $\theta = (\theta_A, \theta_B)$ and $g(w, \theta) = g_1^2(w, \theta_A) + g_1^2(w, \theta_B)$ in Formula (1),
- use the constraints $\theta_A^T \theta_A + \theta_B^T \theta_B = 1$, or two constraints: $\theta_A^T \theta_A = 1$ and $\theta_B^T \theta_B = 1$.

With two constrains, we have two Lagrange multipliers λ_A and λ_B in Formula (2).

Likewise, if the data points are either in plane A or plane B and you want to find these planes based on unlabeled training set observations, proceed exactly as for fitting a line in 3D (the previous paragraph), but this time use $g(w, \theta) = g_1(w, \theta_A) g_1(w, \theta_B)$ instead. By "unlabeled", I mean that you don't know which plane a training set point is assigned to. This is actually an unsupervised clustering problem. The training set points (called cloud) don't have to reside in two separate flat planes: the cloud consists of two sub-clouds A and B, possibly overlapping, each with its own thickness.

This generalizes in various ways: replacing planes by ellipsoids, working in higher dimensions, or with multiple sub-clouds A, B, C and so on. One interesting example is as follows. Training set points are distributed in two clusters A and B, and you want to find the centers of these clusters. Typically, one uses a mixture [Wiki] to model this situation. In our model-free framework, with the convention that w is a row vector and θ_A, θ_B are column vectors, the problem is stated as

$$g(w, \theta) = ||w^T - \theta_A||^{p/2} \cdot ||w^T - \theta_B||^{p/2} \tag{9}$$

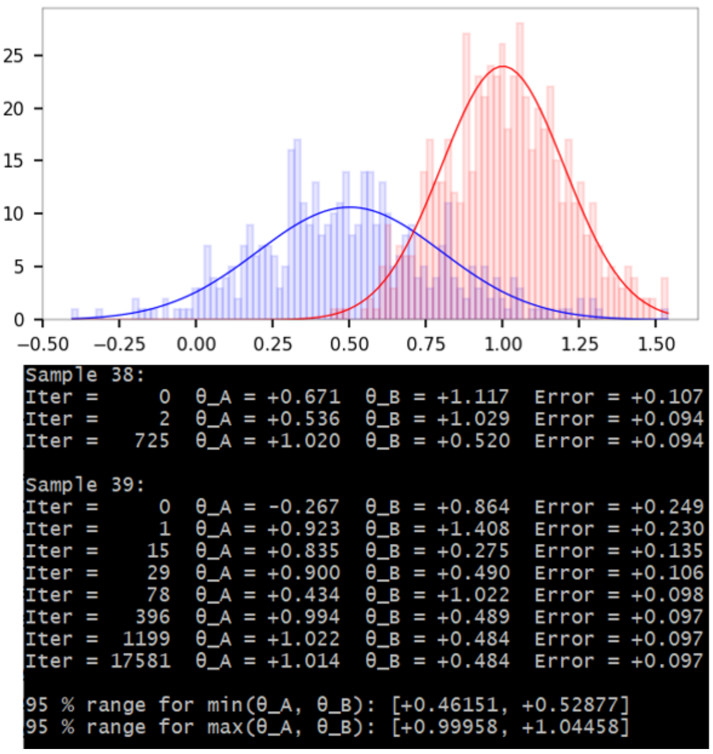
where w, θ_A, θ_B have same dimensions, and there is no constraint on $\theta = (\theta_A, \theta_B)$. Here, $p > 0$ is an hyperparameter [Wiki]. If you use an iterative algorithm to find an optimum solution $\theta^* = (\theta_A^*, \theta_B^*)$, that is, the two centers θ_A^*, θ_B^*, it makes sense to start with $\theta_A = \theta_B$ being the centroid of the whole cloud. The solution may not be unique. Obviously, the problem is symmetric in θ_A and θ_B, but there may be more subtle types of non-uniqueness.

A more general formulation, not discussed here, is to replace $w^T - \theta_A$ and $w^T - \theta_B$ respectively by $\Lambda(w^T - \theta_A)$ and $\Lambda(w^T - \theta_B)$, where Λ is a square invertible matrix, considered and treated as an extra parameter, part of the general parameter $\theta = (\theta_A, \theta_B, \Lambda)$. As a pre-processing step, one can normalize the data, so that its center is the origin and its covariance matrix – after a suitable rotation – is diagonal. My method preserves the norm $|| \cdot ||$, under such transformations.

3.4.1 Example: confidence region for the cluster centers

I tested model (9) in one dimension with $n = 1000$ observations, $p = 1$ and synthetic data generated as a mixture of two normal distributions. The purpose was to identify the cluster centers. The results are pictured in Figure 5. The centers are correctly identified, despite the huge overlap between the two clusters (the purple area in the histogram). The histogram shows the point distribution in cluster A (blue) and B (red), here for the test labeled "Sample 39" in the screenshot.

I computed confidence intervals for the centers, using parametric bootstrap [Wiki]. The theoretical values for the center locations are 0.50 and 1.00. The 95% confidence intervals are $[0.46, 0.53]$ and $[1.00, 1.04]$. The small bias is due to the uneven point counts and variances in the generated clusters: 400 points and $\sigma = 0.3$ in A, versus 600 points and $\sigma = 0.2$ in B.



```
Sample 38:
Iter =      0  θ_A = +0.671  θ_B = +1.117  Error = +0.107
Iter =      2  θ_A = +0.536  θ_B = +1.029  Error = +0.094
Iter =    725  θ_A = +1.020  θ_B = +0.520  Error = +0.094

Sample 39:
Iter =      0  θ_A = -0.267  θ_B = +0.864  Error = +0.249
Iter =      1  θ_A = +0.923  θ_B = +1.408  Error = +0.230
Iter =     15  θ_A = +0.835  θ_B = +0.275  Error = +0.135
Iter =     29  θ_A = +0.900  θ_B = +0.490  Error = +0.106
Iter =     78  θ_A = +0.434  θ_B = +1.022  Error = +0.098
Iter =    396  θ_A = +0.994  θ_B = +0.489  Error = +0.097
Iter =   1199  θ_A = +1.022  θ_B = +0.484  Error = +0.097
Iter =  17581  θ_A = +1.014  θ_B = +0.484  Error = +0.097

95 % range for min(θ_A, θ_B): [+0.46151, +0.52877]
95 % range for max(θ_A, θ_B): [+0.99958, +1.04458]
```

Figure 5: Finding the two centers θ_A^*, θ_B^* in sample 39; $n = 1000$

The bias visible in Figure 6 could be exacerbated by the Mersenne twister pseudo-random number generator [Wiki] used in `numpy.random`, especially in extensive simulations such as this one: see [2]. In this experiment, the twister was called 800 million times. Then, I used the most extreme estimates based on 40 tests, to get the upper and lower bounds of the confidence intervals. This could have contributed to the bias as well, as it is not the most robust approach: running 400 tests and building the confidence intervals based on test percentiles is more robust. But it requires 10 times more computations.

Out of curiosity, I decided to plot the confidence region [Wiki] for (θ_A^*, θ_B^*), this time using 5000 tests, based on one trillion pseudo-random numbers: 5000 tests $\times$ 100,000 iterations per test $\times$ two coordinates. It took about two hours of computing time on my laptop. The result is displayed in Figure 6. Not surprisingly, the confidence region is elliptic: see exercises 27 and 28 in [6] for the explanation.

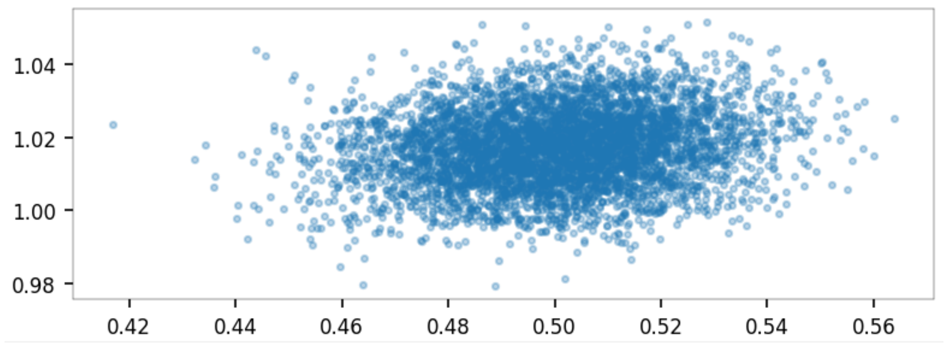


Figure 6: Biased confidence region for (θ_A^*, θ_B^*); same example as in Figure 5; true value is $(0.5, 1.0)$

The implementation details are in the short Python code in section 3.4.4. This unsupervised center-searching algorithm is told that there are two clusters, but it does not know what proportion of points belong to A or B, nor the variances attached to each distribution, or which one is labeled A or B. If the number of clusters is not specified, try different values. I describe a blackbox solution to find the optimum number of clusters, in section 3.4.4 in my book on stochastic processes [6].

3.4.2 Exact solution and caveats

Let W_k be the k-th observation ($k = 1, \dots, n$) stored as a row vector, W the data set (a matrix with n rows) and $\theta = (\theta_A, \theta_B)$ where θ_A, θ_B are column vectors, each with the same dimension as W_k. Then, according to (1), any optimum solution satisfies

$$(\theta_A^*, \theta_B^*) = \arg\min_{\theta} \sum_{k=1}^{n} g^2(W_k, \theta) = \arg\min_{\theta_A, \theta_B} \sum_{k=1}^{n} ||W_k^T - \theta_A||^p \cdot ||W_k^T - \theta_B||^p. \tag{10}$$

As usual, the mean squared error (MSE) is the sum in (10) computed at $\theta^* = (\theta_A^*, \theta_B^*)$, and divided by n. It follows immediately that if h is a distance-preserving mapping (rotation, symmetry or translation) and θ^* is an optimum solution for the data set W, then $h(\theta^*)$ is optimum for $h(W)$, since MSE is invariant under such transformations. Thus, without loss of generality, one can assume that the data set W is centered at the origin.

You can choose a different p for each cluster – say p_A, p_B – or a weighted sum as in Formula (5). If $p = 2$ and there is only one cluster (thus no θ_B), then θ_A^* is the centroid of the point cloud (the W_k's). Now let the clusters be well separated to the point that each observation W_k coincides either with the center of A, or the center of B. Then there is only one unique optimum: θ_A^* is the centroid of one cluster, θ_B^* is the centroid of the other cluster, and the MSE is zero.

If there are three clusters, formula (10) becomes

$$(\theta_A^*, \theta_B^*, \theta_C^*) = \arg\min_{\theta_A, \theta_B, \theta_C} \sum_{k=1}^{n} ||W_k^T - \theta_A||^p \cdot ||W_k^T - \theta_B||^p \cdot ||W_k^T - \theta_C||^p. \tag{11}$$

If $p > 0$ is an even integer (or both p_A, p_B are even integers), then finding the optimum in (10) or (11) consists in solving a system of multivariate polynomials, where the unknowns are the components of θ_A and θ_B. The more clusters, the higher the degrees of the polynomials. In particular, in one dimension with $p = p_A = p_B = 2$, if the data set (the point cloud W) is centered at the origin, then the optimum (θ_A^*, θ_B^*) satisfies

$$\theta_A^* \theta_B^* = -\sigma_W^2, \quad \theta_A^* + \theta_B^* = \frac{1}{n\sigma_W^2} \sum_{k=1}^{n} W_k^3, \quad \text{with } \sigma_W^2 = \frac{1}{n} \sum_{k=1}^{n} W_k^2. \tag{12}$$

Formula (12) can easily be generalized to any dimension. It is obtained by vanishing the gradient to find the minimum in (10). Unfortunately, no exact formula exists for $p = 1$. However, in one dimension for $p = 1$, we have

$$|W_k - \theta_A| \cdot |W_k - \theta_B| = \frac{1}{2} \cdot |(W_k - \theta_A)^2 + (W_k - \theta_B)^2 - (\theta_A - \theta_B)^2|.$$

Based on the few tests done so far in one dimension, in general $p = 1$ works better than $p = 2$. If the two clusters are moderately unbalanced as in Figure 5, then $p = 1$ still does well. However, if they are strongly unbalanced as in Figure 7, the method fails. It is still fixable, by choosing two different p's, denoted as p_A and p_B. Then the optimum corresponds to the larger p attached to the smaller cluster: the blue one, in Figure 7. In this case $p_A = 3, p_B = 1$ works just as well as $p_A = 1, p_B = 1$ does in Figure 5. The blue cluster has 1500 points in Figure 7, the red one 8500. The two centers are 0.5 and 1.0, and the standard deviations are 0.1 and 0.2 respectively for the small and large cluster.

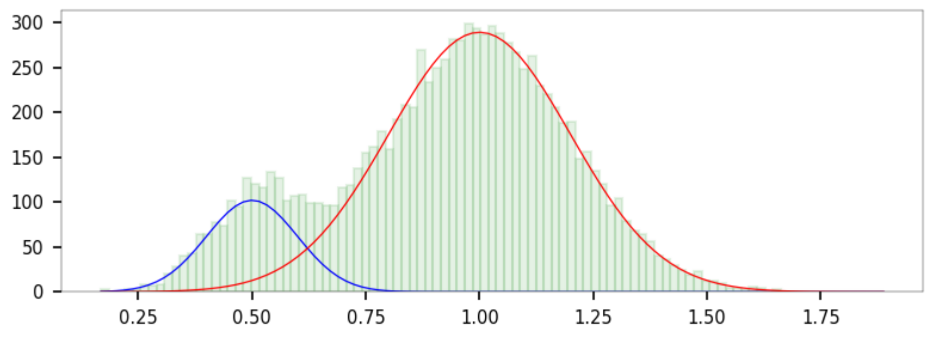


Figure 7: Challenging mixture, requiring $p_A = 3, p_B = 1$ to identify the two cluster centers

It is a good practice to try $(1, 1), (2, 1)$ and $(3, 1)$ for (p_A, p_B), to see which one provides the best fit as illustrated in section 3.4.3. This is particularly useful in blackbox systems, when automatically processing thousands of datasets without a human being ever looking at any of them. Because the "best" solution – from a visual point of view – is sometimes a local rather than a global minimum, I recommend to list all the local minima found during the search for an optimum (θ_A^*, θ_B^*).

Of course, there is a limit to this methodology, as well as to any other classifiers or mode-searching algorithms: if the mixture has just one mode, it is impossible to find two distinct meaningful centers, no matter what method you use. This happens when the cluster centers are truly distinct, but the variances are huge, or if one cluster contains very few points. Another example is when the clusters have irregular, non-convex shapes, with multiple centers and holes. In the latter case, the methodology is still useful to find the local modes.

3.4.3 Comparison with K-means clustering

I included the K-means clustering method [Wiki] in the Python code in section 3.4.4. Here I compare Kmeans with my method, on two datasets, each with $n = 1000$ points and two overlapping clusters. The theoretical cluster centers based on the underlying mixture model are 0.5 and 1.0 respectively. The datasets are pictured in Figure 5 and 7.

On challenging data with significant cluster overlapping, my method frequently outperforms Kmeans. However, on very skewed data, you need two exponents p_A, p_B in Formula (10), rather than just p to get the best performance. When using $(p_A, p_B) = (3, 1)$, my method is denoted as $(3, 1)$. Likewise, with $(p_A, p_B) = (1, 1)$ or $(p_A, p_B) = (2, 1)$, my method is denoted respectively as $(1, 1)$ and $(2, 1)$. Intuitively, model $(3, 1)$ – compared to the default version $(1, 1)$ – allows you to reduce the influence of a highly dominant cluster A, by penalizing its contribution to MSE, with a small p_A. Due to the symmetry of the problem, model $(3, 1)$ and $(1, 3)$ yield the same optimum centers with labels swapped, and the same MSE at the optimum. The reference model with one single cluster coinciding with the centroid of the whole dataset, is called the "base" model. An alternative is to use the medoid [Wiki] rather than the centroid in the base model.

The remaining of this section, besides model comparison, focuses on automatically detecting whether the default model $(1, 1)$ is good enough for a specific dataset, or whether you should use the cluster centers generated by $(2, 1)$ or $(3, 1)$ instead. The decision is based on the MSE defined at the beginning of section 3.4.2. However comparing MSE$(1, 1)$ and MSE$(1, 3)$ even for the same θ and on the same dataset is meaningless. This is the challenge that we face.

To solve this problem, I start by computing MSE$(1, 1)$ and MSE$(3, 1)$ for all methods and both data sets. I skipped MSE$(2, 1)$ as it yields similar solutions to MSE$(3, 1)$. The results are summarized in Table 3 for the first dataset, and Table 4 for the second dataset. The vector $\theta^*(1, 1) = (\theta_A^*(1, 1), \theta_B^*(1, 1))$ contains the two optimum centers according to model $(1, 1)$, given a data set. The same notation $\theta^*(3, 1)$ is used for model $(3, 1)$.

Model	θ_A	θ_B	MSE$(1,1)$	MSE$(3,1)$
$\theta^*(1,1)$	0.53554	1.02221	0.09804	0.02981
$\theta^*(3,1)$	0.20602	0.94284	0.12986	0.02147
Kmeans	0.42525	1.01235	0.10086	0.03049
Base	0.80392	0.80392	0.11673	0.03661

Table 3: MSE for different methods and θs, same data set as in Figure 5

Model	θ_A	θ_B	MSE$(1,1)$	MSE$(3,1)$
$\theta^*(1,1)$	0.72435	1.09296	0.05743	0.01092
$\theta^*(3,1)$	1.06020	0.52378	0.06871	0.00686
Kmeans	0.65856	1.09833	0.05857	0.01340
Base	0.92682	0.92682	0.06812	0.01156

Table 4: MSE for different methods and θs, same data set as in Figure 7

The red entries in Tables 3 and 4 correspond to an optimum for models $(1, 1)$ and $(3, 1)$. The centers for M$(1, 1)$ in the first dataset (Table 3), and for M$(3, 1)$ in the second dataset (Tables 4), are much closer to the true values 0.5 and 1.0 than those produced by Kmeans. However, to claim that my method is better than Kmeans, you need a mechanism to decide when M$(1, 1)$ or M$(3, 1)$ is the best fit. This is still a work in progress. As a starting point, the following arguments provide empirical rules to decide.

- For the first data set, MSE$(1,1)$ evaluated at the centroid of the whole data set (the base model) is better (lower) than when evaluated at $\theta^*(3,1)$, suggesting that $\theta^*(3,1)$ is not a great solution here. Thus the default model $(1,1)$ should be preferred to $(3,1)$ in this case.
- For the second data set, MSE$(1,1)$ evaluated at the centroid is about the same as when evaluated at $\theta^*(3,1)$, suggesting that $\theta^*(3,1)$ is not that bad, at least not as bad as in the previous case. Thus model $(3,1)$ should not automatically be ruled out in this case. It is also an indicator that this data set is more challenging than the previous one.
- For the second data set, MSE$(3,1)$ evaluated at $\theta^*(3,1)$ is better than when evaluated at $\theta^*(1,1)$. This does not mean anything: of course MSE$(3,1)$ is always best at $\theta^*(3,1)$, by design. However the ratio of these two MSE's, $\rho = 0.01092/0.00686 \approx 1.59$, is quite high here. To the contrary, for the first data set $\rho \approx 1.39$ is much smaller. Thus model $(3,1)$ is more justified for the second dataset than for the first one.

Note that he computation of MSE$(1,1)$ or MSE$(3,1)$ is performed without knowing which cluster a point is assigned to. Indeed, point allocation is discussed nowhere in my method: you find the centers without allocating points to specific clusters. Once the two centers θ_A^*, θ_B^* have been computed, each point W_k is assigned to the closest cluster. Proximity is measured as the distance between the point and the cluster center. Then choose the model – $(1,1)$ or $(3,1)$ – minimizing the sum of these distances.

It is my guess that replacing $||W_k - \theta_A||^2$ and $||W_k - \theta_B||^2$ by $||W_k - \theta_A||^{p_A}$ and $||W_k - \theta_B||^{p_B}$ in the Kmeans procedure will yield better results similar to my method. Again, θ_A, θ_B are the two cluster centers, and W_k is the k-th observation. Even $p_A = p_B = 1$ could lead to significant improvements in Kmeans in the presence of outliers (for instance outliers from a large cluster spilling over to a nearby smaller cluster). This approach is somewhat similar to K-medians clustering [Wiki]. Using linear rather than power weights may have the same effect.

Conclusions

My method frequently works better than Kmeans to detect the centers when clusters strongly overlap and are unbalanced. However, this assumes that you have a mechanism to choose between model $(1,1)$ and $(2,1)$ or $(3,1)$. If you know beforehand that your data is highly skewed as in Figure 7, then model $(3,1)$ is a good candidate to begin with. When the clusters are well separated and one or two of the distributions is asymmetric, model $(1,1)$ tends to correctly identify the cluster medians, rather than the standard centers (the mean).

The method is simple and fast: it does not perform point allocation. You can use it to find initial center configurations as first approximations in more complex algorithms, or in the context of "unsupervised" logistic regression to detect the two clusters when there is no independent variable. Exact solutions such as (12) are available in any dimension for two and more clusters, for instance for the $(2,2)$, $(2,2,2)$ and $(4,2)$ models. Other original clustering algorithms are described in my book on stochastic processes [6].

Next steps: test on asymmetric synthetic data modeled as a mixture of normal and gamma distributions with unequal cluster sizes and variances, reduce volatility, investigate the model $(1, \frac{1}{3})$ – the sister of $(3,1)$ – and generalize the method to two or three dimensions and more than two clusters. One of the goals is to identify when my method performs better than Kmeans, to learn more about Kmeans and further improve it.

3.4.4 Python code

The Python code `mixture1D.py` for the one-dimensional case in section 3.4.1, is also on GitHub here. Note that W_A, W_B and W are vectors, respectively with n_A, n_B and n elements. The vector operations (multiplications and so on) are implicitly performed component-wise, without using a loop on the elements. When $p = 2$, `Error` corresponds to the mean squared error. I use color transparency – the parameter `alpha` in the histogram function `plt.hist` – to visualize the overlap between the two components of the Gaussian mixture: see result in left plot, Figure 5.

The optimum θ_A^*, θ_B^* (the cluster centers) are obtained via Monte-Carlo simulations, using 100,000 sampled θ_A, θ_B per test. On the right plot showing convergence to the optimum, you can see that θ_A and θ_B are randomly flipped back and forth within each test. The algorithm senses that there are two distinct centers; however, it can't tell which one is labeled A or B. Afterall, this is unsupervised learning. To address this issue, when computing the confidence intervals, I use the notation θ_A for the center on the left, and θ_B for the other one. That is, $\theta_A < \theta_B$. Finally, I run 40 tests to determine a 95% confidence interval for the optimum values. Results are displayed in the screenshot in Figure 5. Zoom in for a better view.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
from sklearn.cluster import KMeans
```

```
N_tests = 5 # number of data sets being tested
n_A = 1500 # number of points in cluster A
n_B = 8500 # number of points in cluster B
n = n_A + n_B
Ones = np.ones((n)) # array with 1's
p_A = 3
p_B = 1
np.random.seed(438713)
min_θ_A = 99999999
min_θ_B = 99999999
max_θ_A = -99999999
max_θ_B = -99999999
CR_x=[] # confidence region for (best_θ_A, best_θ_A), 1st coordinate
CR_y=[] # confidence region for (best_θ_A, best_θ_A), 2nd coordinate

def compute_MSE(θ_A, θ_B, p_A, p_B, W):
   n = W.size
   MSE = (1/n) * np.sum((abs(W - θ_A * Ones)**p_A) * (abs(W - θ_B * Ones)**p_B))
   return MSE

for sample in range(N_tests): # new dataset at each iteration

   # W_A = np.random.normal(0.5, 2, size=n_A)
   # W_B = 1 + np.random.gamma(8, 5, size=n_B)/4
   W_A = np.random.normal(0.5, 0.1, size=n_A)
   W_B = np.random.normal(1.0, 0.2, size=n_B)
   W   = np.concatenate((W_A, W_B))
   min_MSE=99999999
   print('Sample %1d:' %(sample))

   for iter in range(10000):

      θ_A = np.amin(W) + (np.amax(W)-np.amin(W))*np.random.rand()
      θ_B = np.amin(W) + (np.amax(W)-np.amin(W))*np.random.rand()
      MSE = compute_MSE(θ_A, θ_B, p_A, p_B, W) # MSE for my method
      if MSE < min_MSE:
         min_MSE=MSE
         print('Iter = %5d θ_A = %+8.4f θ_B = %+8.4f MSE = %+12.4f' \
               %(iter,θ_A ,θ_B, MSE))
         best_θ_A = min(θ_A, θ_B)
         best_θ_B = max(θ_A, θ_B)

   if best_θ_A < min_θ_A:
      min_θ_A = best_θ_A
   if best_θ_A > max_θ_A:
      max_θ_A = best_θ_A
   if best_θ_B < min_θ_B:
      min_θ_B = best_θ_B
   if best_θ_B > max_θ_B:
      max_θ_B = best_θ_B
   CR_x.append(best_θ_A)
   CR_y.append(best_θ_B)
   print()

   # get centers from Kmeans method (for comparison purposes)
   V   = W.copy()
   km = KMeans(n_clusters=2)
   km.fit(V.reshape(-1,1))
   centers_kmeans=km.cluster_centers_
   kmeans_A=min(centers_kmeans[0,0],centers_kmeans[1,0])
   kmeans_B=max(centers_kmeans[0,0],centers_kmeans[1,0])

   MSE_kmeans = compute_MSE(centers_kmeans[0,0], centers_kmeans[1,0], p_A, p_B, V)
   centroid=(1/n)*np.sum(W)
   centroid_A=(1/n_A)*np.sum(W_A)
   centroid_B=(1/n_B)*np.sum(W_B)
   median_A=np.median(W_A)
   median_B=np.median(W_B)
   MSE_base = compute_MSE(centroid, centroid, p_A, p_B, W) # MSE for base model
   MSE_tc1 = compute_MSE(centroid_A, centroid_B, p_A, p_B, W)
   MSE_tc2 = compute_MSE(centroid_B, centroid_A, p_A, p_B, W)
   MSE_true_centers = min(MSE_tc1,MSE_tc2)
   MSE_tm1 = compute_MSE(median_A, median_B, p_A, p_B, W)
   MSE_tm2 = compute_MSE(median_B, median_A, p_A, p_B, W)
   MSE_true_medians = min(MSE_tm1,MSE_tm2) # MSE for base model

   print('True centers θ_A = %+8.4f θ_B = %+8.4f MSE = %+12.4f' \
        %(centroid_A,centroid_B,MSE_true_centers))
   print('model (%1d,%1d) θ_A = %+8.4f θ_B = %+8.4f MSE = %+12.4f' \
        %(p_A,p_B,best_θ_A,best_θ_B,min_MSE))
   print('Kmeans   θ_A = %+8.4f θ_B = %+8.4f MSE = %+12.4f' \
        %(kmeans_A,kmeans_B,MSE_kmeans))
   print('True medians θ_A = %+8.4f θ_B = %+8.4f MSE = %+12.4f' \
        %(median_A,median_B,MSE_true_medians))
   print('Base     θ_A = %+8.4f θ_B = %+8.4f MSE = %+12.4f' \
        %(centroid,centroid,MSE_base))
   print()

print('95 %% range for min(θ_A, θ_B): [%+8.4f, %+8.4f]' %(min_θ_A ,max_θ_A))
print('95 %% range for max(θ_A, θ_B): [%+8.4f, %+8.4f]' %(min_θ_B ,max_θ_B))

# intialize plotting parameters
plt.rcParams['axes.linewidth'] = 0.2
plt.rc('axes',edgecolor='black') # border color
plt.rc('xtick', labelsize=7) # font size, x axis
plt.rc('ytick', labelsize=7) # font size, y axis

# plotting histogram and density
bins=np.linspace(min(W), max(W), num=100)
plt.hist(W_A, color = "blue", alpha=0.2, edgecolor='blue',bins=bins)
plt.hist(W_B, color = "red", alpha=0.3, edgecolor='red',bins=bins)
plt.hist(W, color = "green", alpha=0.1, edgecolor='green',bins=bins)
# plt.plot(bins, 8*norm.pdf(bins,0.5,0.3),color='blue',linewidth=0.6)
# plt.plot(bins, 12*norm.pdf(bins,1,0.2),color='red',linewidth=0.6)
plt.show()

# plotting confidence region
if N_tests > 50:
   plt.scatter(CR_x,CR_y,s=6,alpha=0.3)
   plt.show()
```

References

[1] Vincent Granville. Computer vision: Shape classification via explainable AI. *Preprint*, pages 1–7, 2022. MLTechniques.com [Link]. 7, 9

[2] Vincent Granville. Detecting subtle departures from randomness. *Preprint*, pages 1–14, 2022. MLTechniques.com [Link]. 19

[3] Vincent Granville. Interpretable machine learning: Multipurpose, model-free, math-free fuzzy regression. *Preprint*, pages 1–11, 2022. MLTechniques.com [Link]. 5

[4] Vincent Granville. Little known secrets about interpretable machine learning on synthetic data. *Preprint*, pages 1–14, 2022. MLTechniques.com [Link]. 5

[5] Vincent Granville. New perspective on the Riemann Hypothesis. *Preprint*, pages 1–23, 2022. MLTechniques.com [Link]. 15

[6] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective.* MLTechniques.com, 2022. [Link]. 19, 22

[7] Radim Halir and Jan Flusser. Numerically stable direct least squares fitting of ellipses. *Preprint*, pages 1–8, 1998. [Link]. 6, 9

[8] Christian Hill. *Learning Scientific Programming with Python.* Cambridge University Press, 2016. [Link]. 9

[9] Chris Tofallis. Fitting equations to data with the perfect correlation relationship. *Preprint*, pages 1–11, 2015. Hertfordshire Business School Working Paper[Link]. 2

[10] D. Umbach and K.N. Jones. A few methods for fitting circles to data. *IEEE Transactions on Instrumentation and Measurement*, 52(6):1881–1885, 2003. [Link]. 3, 7

[11] D. A. Vaccari and H. K. Wang. Multivariate polynomial regression for identification of chaotic time series. *Mathematical and Computer Modelling of Dynamical Systems*, 13(4):1–19, 2007. [Link]. 6